US007797363B2

(12) **United States Patent**
Hokenek et al.

(10) **Patent No.: US 7,797,363 B2**
(45) **Date of Patent: Sep. 14, 2010**

(54) **PROCESSOR HAVING PARALLEL VECTOR MULTIPLY AND REDUCE OPERATIONS WITH SEQUENTIAL SEMANTICS**

(75) Inventors: **Erdem Hokenek**, Yorktown Heights, NY (US); **Michael J. Schulte**, Madison, WI (US); **Mayan Moudgill**, White Plains, NY (US); **C. John Glossner**, Carmel, NY (US)

(73) Assignee: **Sandbridge Technologies, Inc.**, White Plains, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 912 days.

(21) Appl. No.: **11/096,921**

(22) Filed: **Apr. 1, 2005**
(Under 37 CFR 1.47)

(65) **Prior Publication Data**

US 2006/0041610 A1 Feb. 23, 2006

Related U.S. Application Data

(63) Continuation-in-part of application No. 10/841,261, filed on May 7, 2004, now Pat. No. 7,593,978.

(60) Provisional application No. 60/560,198, filed on Apr. 7, 2004.

(51) **Int. Cl.**
G06F 15/00 (2006.01)

(52) **U.S. Cl.** **708/603**

(58) **Field of Classification Search** 708/523, 708/603

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,889,689	A	3/1999	Alidina et al.
5,991,785	A	11/1999	Alidina et al.
6,530,010	B1	3/2003	Hung et al.
6,687,724	B1	2/2004	Mogi et al.
6,842,848	B2	1/2005	Hokenek et al.
2003/0120901	A1	6/2003	Hokenek et al.
2004/0073772	A1	4/2004	Hokenek et al.
2004/0073779	A1	4/2004	Hokenek et al.
2005/0071413	A1	3/2005	Schulte et al.

FOREIGN PATENT DOCUMENTS

EP	1623307	A	2/2006
GB	2389433	A	12/2003

(Continued)

OTHER PUBLICATIONS

Schulte et al. "Parallel Saturating Multioperand Adders" 2000, pp. 172-179.*

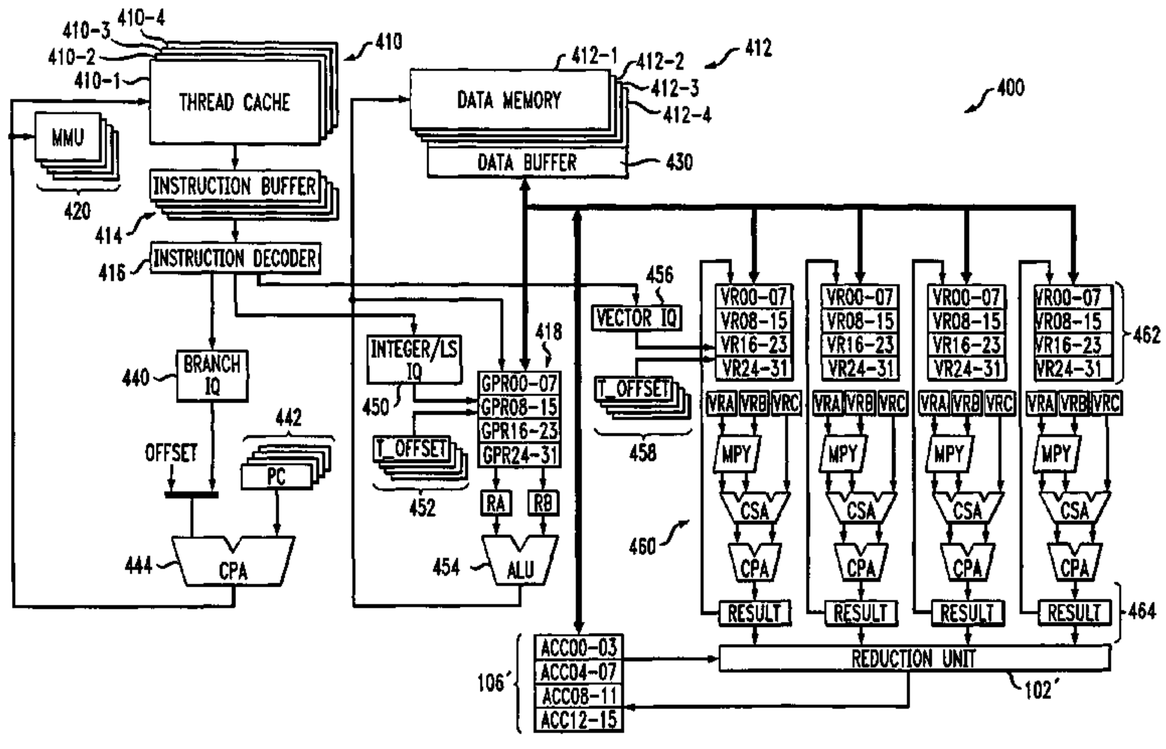
(Continued)

Primary Examiner—Chuong D Ngo
(74) *Attorney, Agent, or Firm*—Lowenstein Sandler PC

(57) **ABSTRACT**

A processor comprises a plurality of arithmetic units, an accumulator unit, and a reduction unit coupled between the plurality of arithmetic units and the accumulator unit. The reduction unit receives products of vector elements from the arithmetic units and a first accumulator value from the accumulator unit, and processes the products and the first accumulator value to generate a second accumulator value for delivery to the accumulator unit. The processor implements a plurality of vector multiply and reduce operations having guaranteed sequential semantics, that is, operations which guarantee that the computational result will be the same as that which would be produced using a corresponding sequence of individual instructions.

21 Claims, 5 Drawing Sheets



FOREIGN PATENT DOCUMENTS

WO 2004/103056 A 12/2004

OTHER PUBLICATIONS

K. Diefendorff et al., "AltiVec Extension to PowerPC Accelerates Media Processing," IEEE Micro, vol. 20, No. 2, pp. 85-95, Mar. 2000.

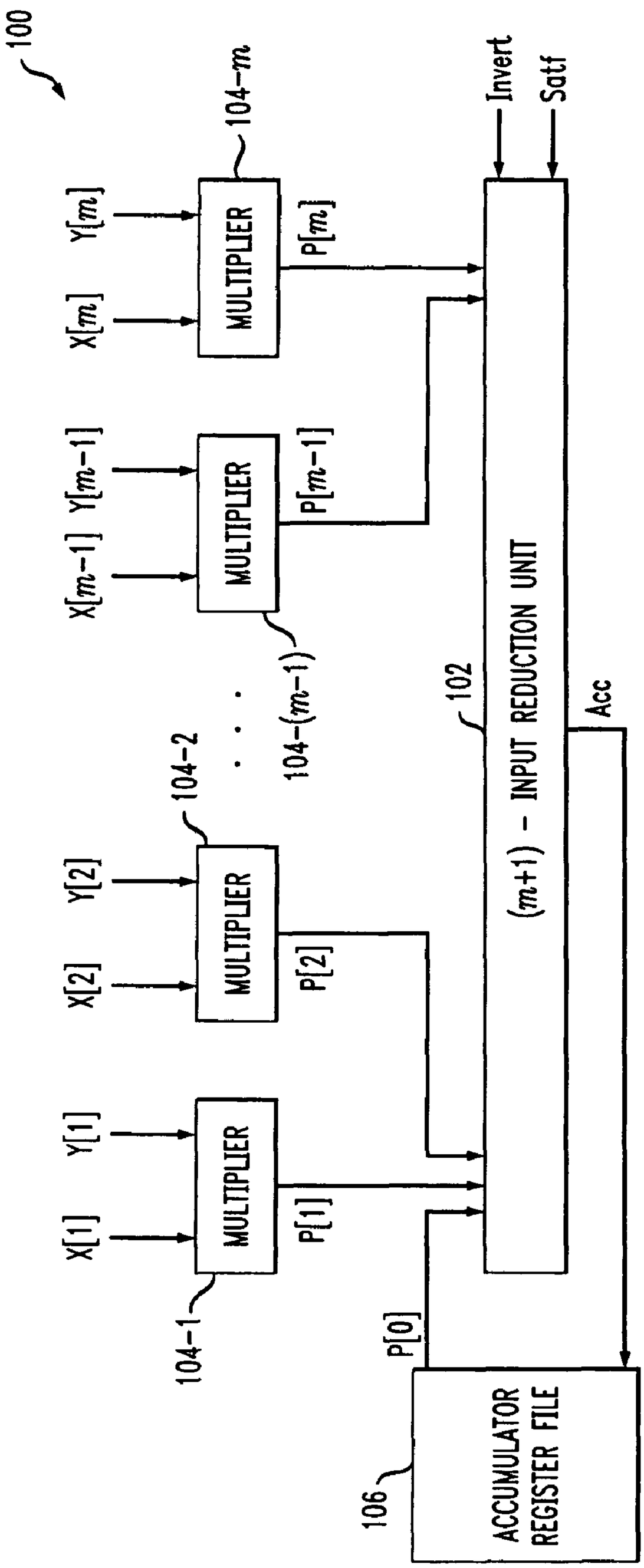
A. Peleg et al., "MMX Technology Extension to the Intel Architecture," IEEE Micro, vol. 16, No. 4, pp. 42-50, 1996.

Supplementary European Search Report dated Sep. 4, 2008.

Peleg et al: "MMX Technology Extension to the Intel Architecture". IEEE Micro, IEEE Service Center, vol. 16, No. 4, Aug. 1, 1996, pp. 42-50.

* cited by examiner

FIG. 1


100
104-m
Y[m]
X[m]
MULTIPLIER
P[m]
Invert
Satf
X[m-1]
Y[m-1]
MULTIPLIER
P[m-1]
104-(m-1)
X[2]
Y[2]
104-2
MULTIPLIER
P[2]
102
(m+1) - INPUT REDUCTION UNIT
Acc
X[1]
Y[1]
MULTIPLIER
P[1]
104-1
P[0]
106
ACCUMULATOR REGISTER FILE

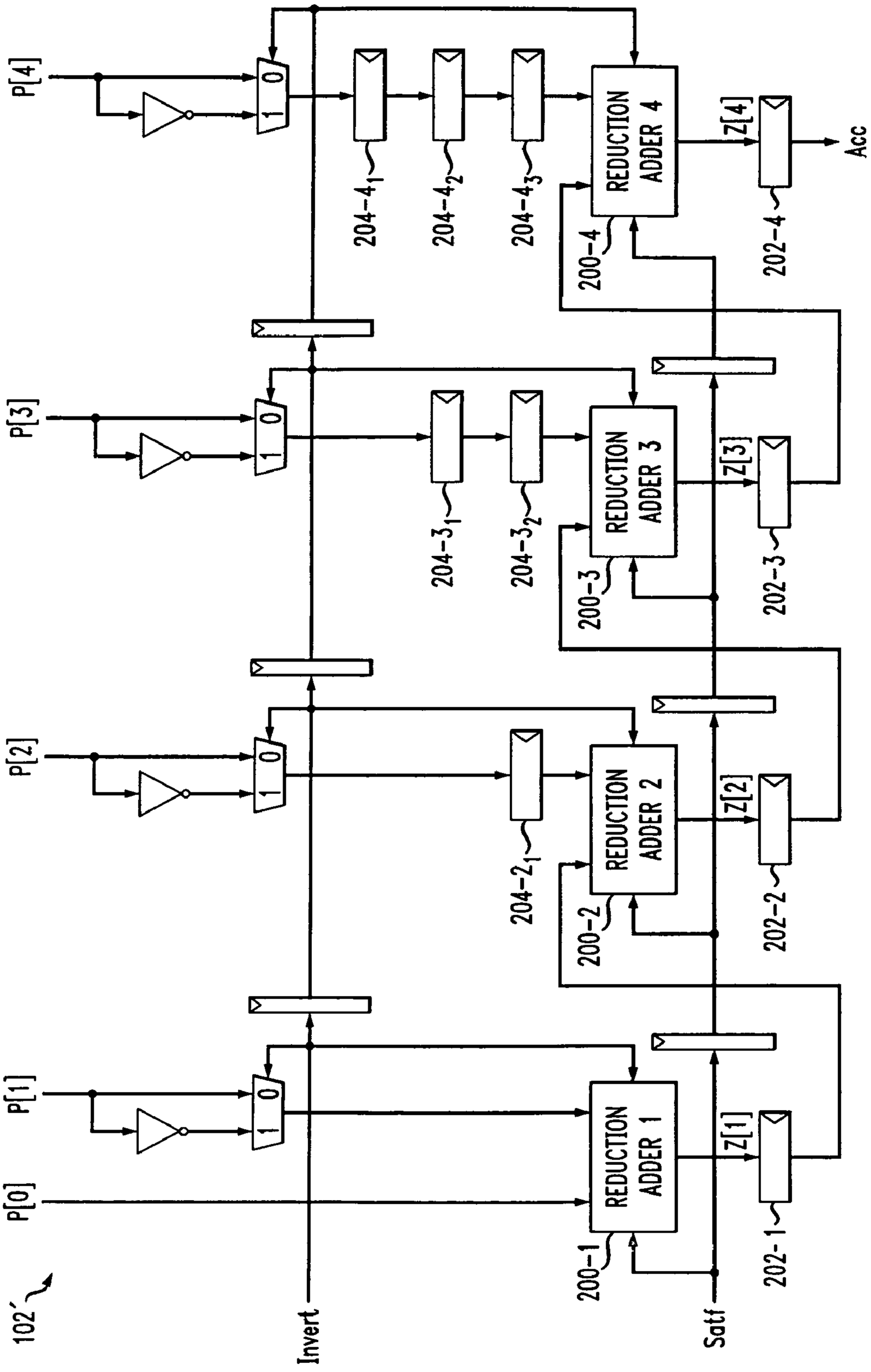

P[4]
1
0
204-4$_1$
204-4$_2$
204-4$_3$
REDUCTION ADDER 4
200-4
Z[4]
202-4
Acc
P[3]
204-3$_1$
204-3$_2$
REDUCTION ADDER 3
200-3
Z[3]
202-3
P[2]
204-2$_1$
REDUCTION ADDER 2
200-2
Z[2]
202-2
P[1]
P[0]
REDUCTION ADDER 1
200-1
Z[1]
202-1
Invert
Satf
102'

FIG. 2

FIG. 3

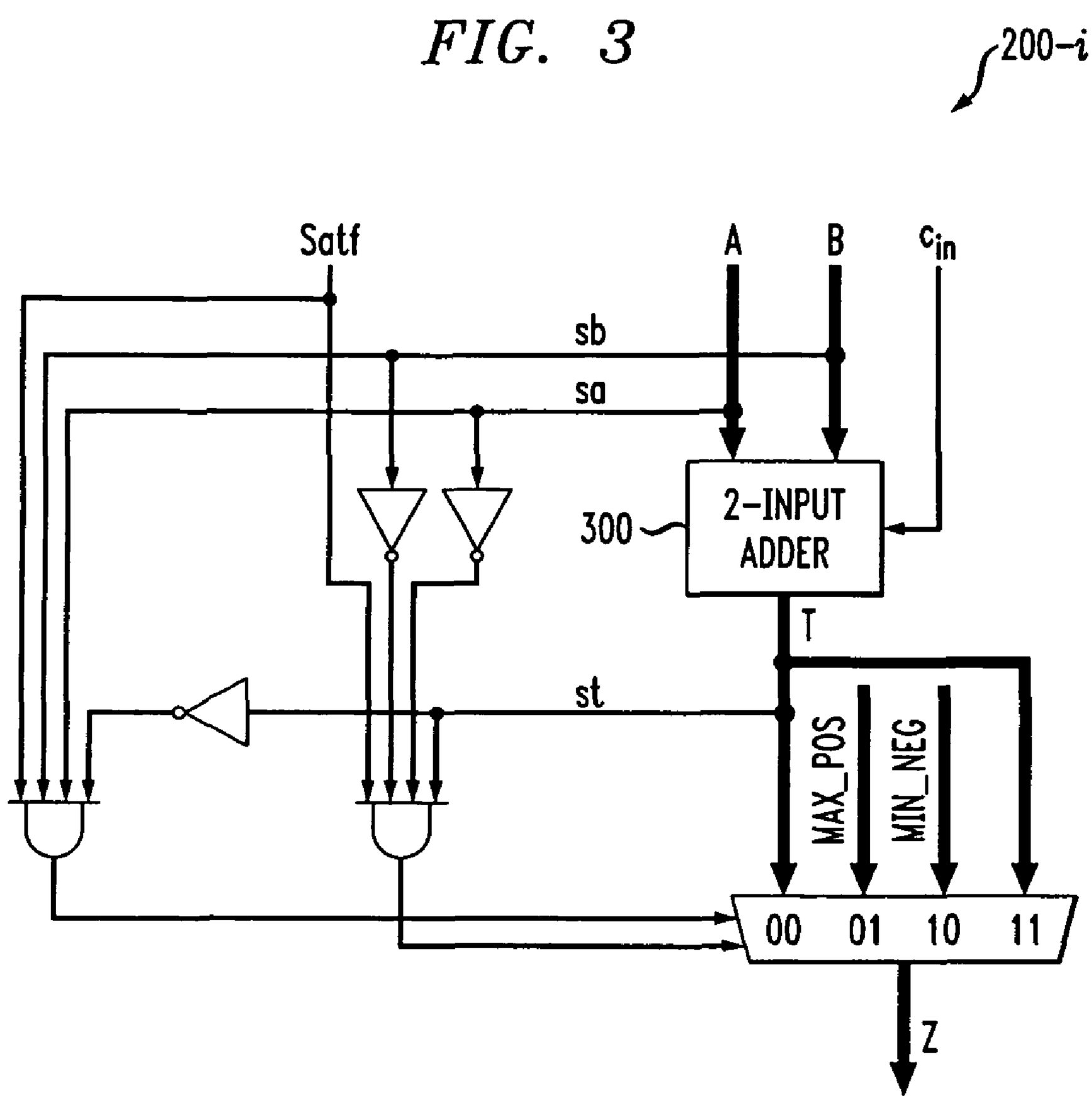

200-i
Satf
A
B
c_{in}
sb
sa
300
2-INPUT ADDER
T
st
MAX_POS
MIN_NEG
00
01
10
11
Z

FIG. 4

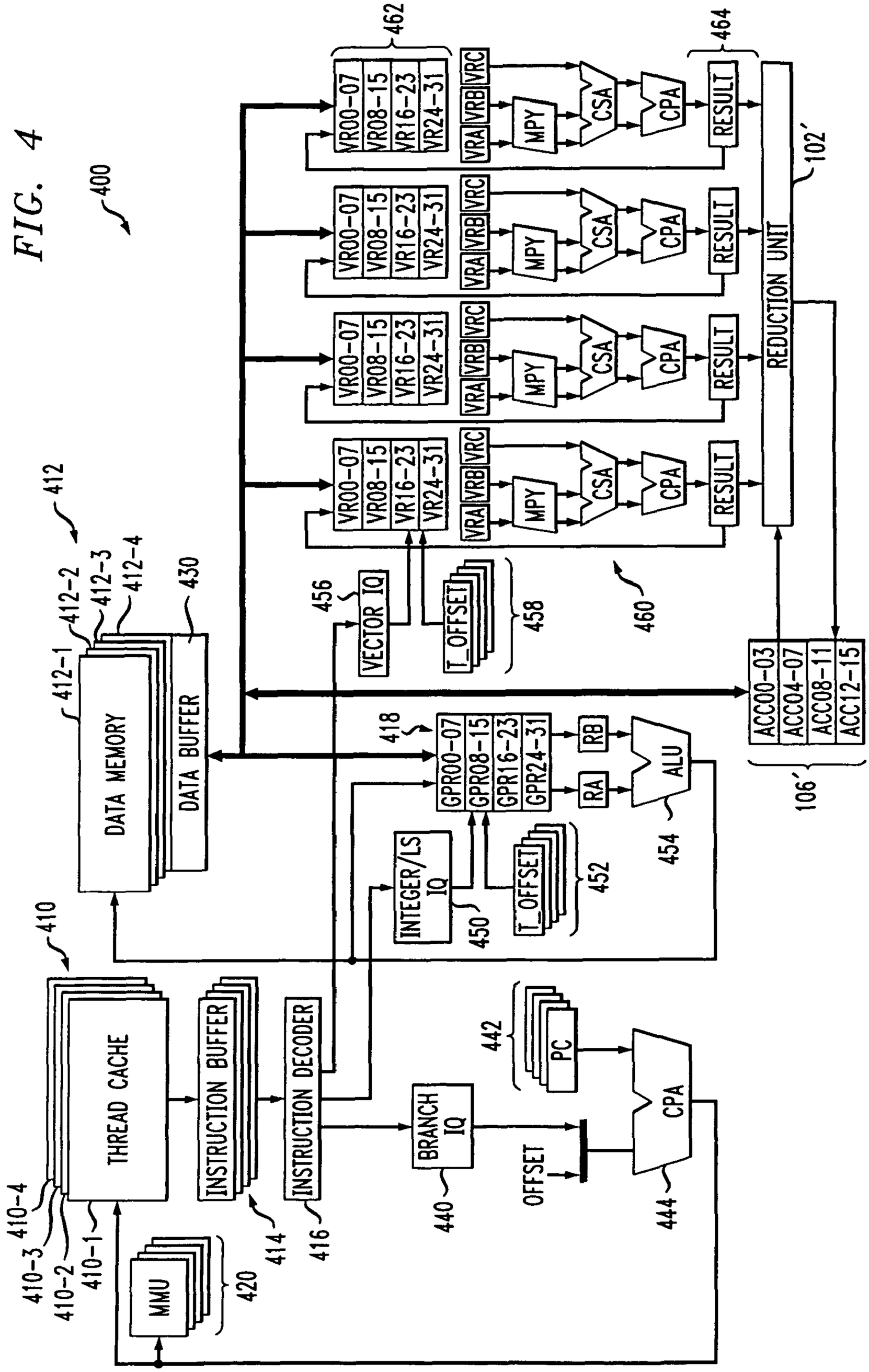

400
410
410-1
410-2
410-3
410-4
412
412-1
412-2
412-3
412-4
DATA MEMORY
DATA BUFFER
430
THREAD CACHE
INSTRUCTION BUFFER
414
INSTRUCTION DECODER
416
MMU
420
418
GPR00-07
GPR08-15
GPR16-23
GPR24-31
RA
RB
ALU
454
INTEGER/LS IQ
450
T_OFFSET
452
VECTOR IQ
456
T_OFFSET
458
460
462
VR00-07
VR08-15
VR16-23
VR24-31
VRA
VRB
VRC
MPY
CSA
CPA
464
RESULT
REDUCTION UNIT
102'
ACC00-03
ACC04-07
ACC08-11
ACC12-15
106'
BRANCH IQ
440
OFFSET
442
PC
444

FIG. 5

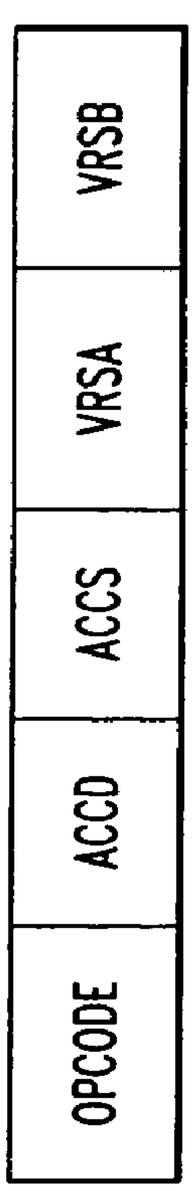
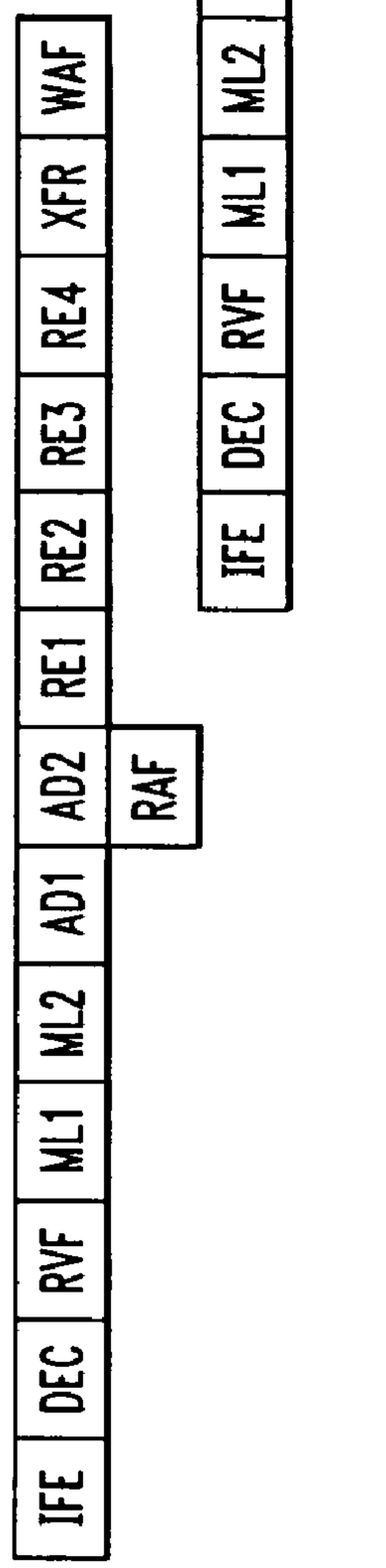

OPCODE
ACCD
ACCS
VRSA
VRSB

FIG. 6


IFE
DEC
RVF
ML1
ML2
AD1
RE1
RE2
RE3
RE4
XFR
WAF
IFE
DEC
RVF
ML1
ML2
AD2
AD3
RE1
RAF
RAF

PROCESSOR HAVING PARALLEL VECTOR MULTIPLY AND REDUCE OPERATIONS WITH SEQUENTIAL SEMANTICS

RELATED APPLICATION(S)

The present application claims the priority of U.S. Provisional Application Ser. No. 60/560,198, filed Apr. 7, 2004 and entitled "Parallel Vector Multiply and Reduce Operations with Sequential Semantics," which is incorporated by reference herein.

The present application is a continuation-in-part of U.S. patent application Ser. No. 10/841,261, filed May 7, 2004 and entitled "Processor Reduction Unit for Accumulation of Multiple Operands With or Without Saturation," now U.S. Pat. No. 7,593,978 which is incorporated by reference herein.

FIELD OF THE INVENTION

The present invention relates generally to the field of digital data processors, and more particularly to arithmetic processing operations and associated processing circuitry for use in a digital signal processor (DSP) or other type of digital data processor.

BACKGROUND OF THE INVENTION

Many digital data processors, including most DSPs and multimedia processors, use binary fixed-point arithmetic, in which operations are performed on integers, fractions, or mixed numbers in unsigned or two's complement binary format. DSP and multimedia applications often require that the processor be configured to perform saturating arithmetic or wrap-around arithmetic on binary numbers.

In saturating arithmetic, computation results that are too large to be represented in a specified number format are saturated to the most positive or most negative number. When a result is too large to represent, overflow occurs. For example, in a decimal number system with 3-digit unsigned numbers, the addition 733+444 produces a saturated result of 999, since the true result of 1177 cannot be represented with just three decimal digits. The saturated result, 999, corresponds to the most positive number that can be represented with three decimal digits. Saturation is useful because it reduces the errors that occur when results cannot be correctly represented, and it preserves sign information.

In wrap-around arithmetic, results that overflow are wrapped around, such that any digits that cannot fit into the specified number representation are simply discarded. For example, in a decimal number system with 3-digit unsigned numbers, the addition 733+444 produces a wrap-around result of 177. Since the true result of 1177 is too large to represent, the leading 1 is discarded and a result of 177 is produced. Wrap-around arithmetic is useful because, if the true final result of several wrap-around operations can be represented in the specified format, the final result will be correct, even if intermediate operations overflow.

As indicated above, saturating arithmetic and wrap-around arithmetic are often utilized in binary number systems. For example, in a two's complement fractional number system with 4-bit numbers, the two's complement addition 0.101+0.100 (0.625+0.500) produces a saturated result of 0.111 (0.875), which corresponds to the most positive two's complement number that can be represented with four bits. If wrap-around arithmetic is used, the two's complement addition 0.101+0.100 (0.625+0.500), produces the result 1.001 (−0.875).

Additional details regarding these and other conventional aspects of digital data processor arithmetic can be found in, for example, B. Parhami, "Computer Arithmetic: Algorithms and Hardware Designs," Oxford University Press, New York, 2000 (ISBN 0-19-512583-5), which is incorporated by reference herein.

Since DSP and multimedia applications typically require both saturating arithmetic and wrap-around arithmetic, it is useful for a given processor to support both of these types of arithmetic.

The above-cited U.S. patent application Ser. No. 10/841,261 discloses an efficient mechanism for controllable selection of saturating or wrap-around arithmetic in a digital data processor.

It may also be desirable in many applications to configure a given DSP, multimedia processor or other type of digital data processor for the performance of dot products or other types of vector multiply and reduce operations. Such operations frequently occur in digital signal processing and multimedia applications. By way of example, second and third generation cellular telephones that support GSM (Global System for Mobile communications) or EDGE (Enhanced Data rates for Global Evolution) standards make extensive use of vector multiply and reduce operations, usually with saturation after each individual multiplication and addition. However, since saturating addition is not associative, the individual multiplications and additions needed for the vector multiply and reduce operation are typically performed in sequential order using respective individual instructions, which reduces performance and increases code size.

A number of techniques have been proposed to facilitate vector multiply and reduce operations in a digital data processor. These include, for example, the parallel multiply add (PMADD) operation provided in MMX technology, as described in A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," IEEE Micro, Vol. 16, No. 4, pp. 42-50, 1996, and the multiply-sum (VMSUM) operation in Altivec Technology, as described in K. Diefendorff et al., "AltiVec Extension to PowerPC Accelerates Media Processing," IEEE Micro, Vol. 20, No. 2, pp. 85-95, March 2000. These operations, however, fail to provide the full range of functionality that is desirable in DSP and multimedia processors. Moreover, these operations do not guarantee sequential semantics, that is, do not guarantee that the computational result will be the same as that which would be produced using a corresponding sequence of individual multiplication and addition instructions.

Accordingly, techniques are needed which can provide improved vector multiply and reduce operations, with guaranteed sequential semantics, in a digital data processor.

SUMMARY OF THE INVENTION

The present invention in accordance with one aspect thereof provides a processor having a plurality of arithmetic units, an accumulator unit, and a reduction unit coupled between the plurality of arithmetic units and the accumulator unit. The reduction unit receives products of vector elements from the arithmetic units and a first accumulator value from the accumulator unit, and processes the products and the first accumulator value to generate a second accumulator value for delivery to the accumulator unit. The processor implements a plurality of vector multiply and reduce operations having guaranteed sequential semantics, that is, operations which guarantee that the computational result will be the same as that which would be produced using a corresponding sequence of individual instructions.

In an illustrative embodiment, the plurality of vector multiply and reduce operations comprises the following set of operations:

1. A vector multiply and reduce add with wrap-around which multiplies pairs of vector elements and adds the resulting products to the first accumulator value in sequential order with wrap-around arithmetic.

2. A vector multiply and reduce add with saturation which multiplies pairs of vector elements and adds the resulting products to the first accumulator value in sequential order with saturation after each multiplication and each addition.

3. A vector multiply and reduce subtract with wrap-around which multiplies pairs of vector elements and subtracts the resulting products from the first accumulator value in sequential order with wrap-around arithmetic.

4. A vector multiply and reduce subtract with saturation which multiplies pairs of vector elements and subtracts the resulting products from the first accumulator value in sequential order with saturation after each multiplication and each subtraction.

The illustrative embodiment advantageously overcomes the drawbacks associated with conventional vector multiply and reduce operations, by providing a wider range of functionality, particularly in DSP and multimedia processor applications, while also ensuring sequential semantics.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** shows a portion of an exemplary processor suitable for use in performing parallel vector multiply and reduce operations in accordance with an illustrative embodiment of the invention.

FIG. **2** shows a more detailed view of the FIG. **1** reduction unit as implemented for a case of m=4 in the illustrative embodiment.

FIG. **3** shows a more detailed view of a reduction adder utilized in the FIG. **2** reduction unit.

FIG. **4** shows an example of a multithreaded processor incorporating the FIG. **2** reduction unit.

FIG. **5** shows an exemplary format for a vector-reduce instruction suitable for execution in the FIG. **4** multithreaded processor.

FIG. **6** illustrates pipelined execution of two vector-reduce instructions from the same thread, utilizing an instruction format of the type shown in FIG. **5**.

DETAILED DESCRIPTION OF THE INVENTION

The present invention will be described in the context of an exemplary reduction unit, accumulator unit, and arithmetic units, and a multithreaded processor which incorporates such units. It should be understood, however, that the invention does not require the particular arrangements shown, and can be implemented using other types of digital data processors and associated processing circuitry.

A given processor as described herein may be implemented in the form of one or more integrated circuits.

FIG. **1** shows a portion of a processor **100** configured in accordance with an illustrative embodiment of the invention. The processor **100** includes an (m+1)-input reduction unit **102** coupled between m parallel multipliers, denoted **104-1**, **104-2**, . . . **104-***m*, and an accumulator register file **106**. The operation of the processor **100** will initially be described below in the context of the computation of a dot product, but the processor may be used to perform other types of parallel vector multiply and reduce operations.

Each of the multipliers **104-***i* computes P[i]=X[i]*Y[i], 1≦i≦m, with or without saturation. The m multiplier outputs are then fed as input operands to the (m+1)-input reduction unit **102**, along with an accumulator value, denoted P[0], from the accumulator register file **106**. The reduction unit **102** computes

$$Acc=P[0]+P[1]+P[2]+\ldots+P[m],$$

where P[0] is set to zero for an initial iteration. In the next iteration, m new elements of X and Y are multiplied, and P[0] is set to the accumulator value, Acc, from the previous iteration. This process continues until the entire dot product is computed. Thus, a k-element dot product can be computed using [k/m] iterations, where each iteration includes m parallel multiplies and an (m+1)-input addition. When used in a saturation mode, the reduction unit performs saturation after each addition, and each multiplier saturates its result when overflow occurs.

The accumulator register file **106** may be viewed as an example of what is more generally referred to herein as an "accumulator unit." Other types of accumulator units may be used in alternative embodiments, as will be appreciated by those skilled in the art. Moreover, the term "unit" as used herein is intended to be construed generally, such that elements of a given unit may but need not be co-located with one another or otherwise have a particular physical relationship to one another. For example, elements of a given unit could be distributed throughout an integrated circuit, rather than co-located at one site in such a circuit.

The accumulator register file **106** can be used to store intermediate accumulator values, which is especially useful in a multi-threaded processor implementation, in which several dot products from individual threads may be computed simultaneously.

The reduction unit **102** in the illustrative embodiment of FIG. **1** also receives two 1-bit control signal inputs, Invert and Satf. When Invert is high, the input operands to the reduction unit are inverted, so that the unit computes

$$Acc=P[0]-P[1]-P[2]-\ldots-P[m].$$

This inverted addition is also referred to herein as subtraction, but is generally considered a type of addition, as will be appreciated by those skilled in the art. When Invert is low, the input operands to the reduction unit are not inverted, so the unit computes

$$Acc=P[0]+P[1]+P[2]+\ldots+P[m].$$

When Satf is high, the reduction unit is in saturation mode. This means that after each intermediate addition in the reduction unit a check is made to determine if the result has incurred overflows. If it has, the result is saturated to the most positive or most negative number in the specified format. When Satf is low, the reduction unit is in wrap-around mode, which means that results that overflow are not saturated.

The use of multipliers **104** in the illustrative embodiment is by way of example only. Other embodiments may use, for example, multiply-accumulate (MAC) units. The term "multiplier" as used herein is intended to include an arithmetic unit, such as a MAC unit, which performs multiplication as well as one or more other functions.

FIG. **2** shows an exemplary reduction unit **102'** suitable for use in the processor **100** and more specifically configured for the case of m=4. This reduction unit is operative to sum four input operands, P[1] to P[4], plus an accumulator value, P[0]. Although the figure shows an (m+1)-input reduction unit for

the specific case of m=4, the design can easily be extended to other values of m, as will be apparent to those skilled in the art.

The reduction unit **102'** uses four 2-input reduction adders, denoted **200-1**, **200-2**, **200-3** and **200-4**, which are connected in series as shown. Each reduction adder is able to add its input operands with or without saturation. The term "reduction adder" as used herein is intended to include, by way of example, a saturating adder.

The first reduction adder **200-1**, also identified as Reduction Adder 1, takes operands P[0] and P[1], and adds them to produce Z[1]=P[0]+P[1], when the input control signal Invert is low. Each remaining reduction adder **200**-(*i*+1), also identified as Reduction Adder i+1, takes two input operands, Z[i] and P[i+1], and adds them to produce a sum, Z[i+1]=Z[i]+P[i+1], when the input control signal Invert is low. Thus, when Invert is low, the output of the reduction unit is

$$Acc=Z[4]=P[0]+P[1]+P[2]+P[3]+P[4].$$

When the input control signal Invert is high, the second input to each reduction adder is bit-wise inverted and the carry-input to each reduction adder is set to one. This causes Reduction Adder 1 to compute Z[1]=P[0]−P[1] and the remaining reduction adders to compute Z[i+1]=Z[i]−P[i+1]. In this case, the output of the reduction unit is

$$Acc=Z[4]=P[0]-P[1]-P[2]-P[3]-P[4].$$

When the input control signal Satf is high, the result of each addition (or subtraction) is saturated when overflow occurs. When Satf is low, the result of each addition (or subtraction) is wrapped around.

The reduction unit **102'** is pipelined to decrease its worst case delay. More specifically, the reduction unit **102'** uses a four-stage pipeline to perform four additions (or four subtractions), where the result of each intermediate addition (or subtraction), Z[i], is stored in a pipeline register **202**-*i*. To have the P[i] operands arrive at the same time as the corresponding Z[i−1] operands, the P[i] operand into Reduction Adder i passes through (i−1) pipeline registers **204**. Thus, operand P[1] passes through no pipeline registers **204**, operand P[2] passes through one pipeline register **204-2**$_1$, operand P[3] passes through two pipeline registers **204-3**$_1$ and **204-3**$_2$, and operand P[4] passes through three pipeline registers **204-4**$_1$, **204-4**$_2$ and **204-4**$_3$, in reaching their respective reduction adders.

FIG. **3** shows one possible implementation of a given one of the reduction adders **200**-*i* of the reduction unit **102'**. The reduction adder **200**-*i* uses a 2-input adder **300** to add two input operands, A and B, plus a carry-in bit, c_{in}, to compute $T=A+B+c_{in}$. If Satf and the signs of A and B, sa and sb, are high, and the sign of the temporary result, st, is low, the output, Z, is saturated to the most negative number in the specified number format, such that Z=MIN_NEG. If Satf and st are high and sa and sb are low, Z is saturated to the most positive number in the specified number format, such that Z=MAX_POS. In all other cases the result from the adder **300** is used as the result, such that Z=T.

It should be understood that the particular reduction adder design shown in FIG. **3** is presented by way of illustrative example only. Numerous alternative reduction adder designs may be used, and the particular adder selected for use in a given implementation may vary based on application-specific factors such as the format of the input operands.

In the pipelined reduction unit, it is possible for m elements of a dot product to be accumulated every clock cycle, through the use of multithreading as described below.

It should be noted that, in non-multithreaded processor implementations, pipelining the reduction unit can cause a large increase in the number of cycles needed to compute each dot product. For example, using a conventional m-stage pipeline without multithreading increases the number of cycles to compute each dot product by roughly a factor of m.

The illustrative embodiment of the present invention addresses this issue by utilizing an approach known as token triggered threading. Token triggered threading is described in U.S. Pat. No. 6,842,848, which is commonly assigned herewith and incorporated by reference herein. The token triggered threading typically assigns different tokens to each of a plurality of threads of a multithreaded processor. For example, the token triggered threading may utilize a token to identify in association with a current processor clock cycle a particular one of the threads of the processor that will be permitted to issue an instruction for a subsequent clock cycle. Although token triggered threading is used in the illustrative embodiment, the invention does not require this particular type of multithreading, and other types of multithreading techniques can be used.

In the illustrative embodiment, the above-noted increase in cycle count attributable to pipelining may be effectively hidden by the processing of other threads, since the multiplications and reductions for one dot product are executed concurrently with operations from other threads. In order to completely hide the increase in cycle count by concurrent execution of threads, the number of cycles between execution of instructions from a given thread should be greater than or equal to the number of pipeline stages in the reduction unit plus any additional cycles needed to write to and read from the accumulator register file **106**.

As indicated previously, the present invention can be advantageously implemented in a multithreaded processor. A more particular example of a multithreaded processor in which the invention may be implemented is described in U.S. Pat. No. 6,968,445 (hereinafter "the '445 Patent"), which is commonly assigned herewith and incorporated by reference herein. This multithreaded processor may be configured to execute RISC-based control code, DSP code, Java code and network processing code. It includes a single instruction multiple data (SIMD) vector processing unit, a reduction unit, and long instruction word (LIW) compounded instruction execution. Examples of threading and pipelining techniques suitable for use with this exemplary multithreaded processor are described in the above-cited '445 Patent.

The reduction unit **102** or **102'** as described herein may be utilized as the reduction unit in such a multithreaded processor, as will be illustrated in conjunction with FIG. **4**. Of course, the invention can be implemented in other multithreaded processors, or more generally other types of digital data processors.

FIG. **4** shows an example of a multithreaded processor **400** incorporating the FIG. **2** reduction unit **102'**. The processor **400** is generally similar to that described in the '445 Patent, but incorporates reduction unit **102'** and accumulator register file **106'** configured as described herein.

The multithreaded processor **400** includes, among other elements, a multithreaded cache memory **410**, a multithreaded data memory **412**, an instruction buffer **414**, an instruction decoder **416**, a register file **418**, and a memory management unit (MMU) **420**. The multithreaded cache **410** includes a plurality of thread caches **410-1**, **410-2**, . . . **410**-N, where N generally denotes the number of threads supported by the multithreaded processor **400**, and in this particular example is given by N=4. Of course, other values of N may be used, as will be readily apparent to those skilled in the art.

Each thread thus has a corresponding thread cache associated therewith in the multithreaded cache **410**. Similarly, the data memory **412** includes N distinct data memory instances, denoted data memories **412-1**, **412-2**, . . . **412**-N as shown.

The multithreaded cache **410** interfaces with a main memory (not shown) external to the processor **400** via the MMU **420**. The MMU **420**, like the cache **410**, includes a separate instance for the each of the N threads supported by the processor. The MMU **420** ensures that the appropriate instructions from main memory are loaded into the multithreaded cache **410**.

The data memory **412** is also typically directly connected to the above-noted external main memory, although this connection is also not explicitly shown in the figure. Also associated with the data memory **412** is a data buffer **430**.

In general, the multithreaded cache **410** is used to store instructions to be executed by the multithreaded processor **400**, while the data memory **412** stores data that is operated on by the instructions. Instructions are fetched from the multithreaded cache **410** by the instruction decoder **416** and decoded. Depending upon the instruction type, the instruction decoder **416** may forward a given instruction or associated information to various other units within the processor, as will be described below.

The processor **400** includes a branch instruction queue (IQ) **440** and program counter (PC) registers **442**. The program counter registers **442** include one instance for each of the threads. The branch instruction queue **440** receives instructions from the instruction decoder **416**, and in conjunction with the program counter registers **442** provides input to an adder block **444**, which illustratively comprises a carry-propagate adder (CPA). Elements **440**, **442** and **444** collectively comprise a branch unit of the processor **400**. Although not shown in the figure, auxiliary registers may also be included in the processor **400**.

The register file **418** provides temporary storage of integer results. Instructions forwarded from the instruction decoder **416** to an integer instruction queue (IQ) **450** are decoded and the proper hardware thread unit is selected through the use of an offset unit **452** which is shown as including a separate instance for each of the threads. The offset unit **452** inserts explicit bits into register file addresses so that independent thread data is not corrupted. For a given thread, these explicit bits may comprise, e.g., a corresponding thread identifier.

As shown in the figure, the register file **418** is coupled to input registers RA and RB, the outputs of which are coupled to an ALU block **454**, which may comprise an adder. The input registers RA and RB are used in implementing instruction pipelining. The output of the ALU block **454** is coupled to the data memory **412**.

The register file **418**, integer instruction queue **450**, offset unit **452**, elements RA and RB, and ALU block **454** collectively comprise an exemplary integer unit.

Instruction types executable in the processor **400** include Branch, Load, Store, Integer and Vector/SIMD instruction types. If a given instruction does not specify a Branch, Load, Store or Integer operation, it is a Vector/SIMD instruction. Other instruction types can also or alternatively be used. The Integer and Vector/SIMD instruction types are examples of what are more generally referred to herein as integer and vector instruction types, respectively.

A vector IQ **456** receives Vector/SIMD instructions forwarded from the instruction decoder **416**. A corresponding offset unit **458**, shown as including a separate instance for each of the threads, serves to insert the appropriate bits to ensure that independent thread data is not corrupted.

A vector unit **460** of the processor **400** is separated into N distinct parallel portions, and includes a vector file **462** which is similarly divided. The vector file **462** includes thirty-two registers, denoted VR**00** through VR**31**. The vector file **462** serves substantially the same purpose as the register file **418** except that the former operates on Vector/SIMD instruction types.

The vector unit **460** illustratively comprises the vector instruction queue **456**, the offset unit **458**, the vector file **462**, and the arithmetic and storage elements associated therewith.

The operation of the vector unit **460** is as follows. A Vector/SIMD block encoded either as a fractional or integer data type is read from the vector file **462** and is stored into architecturally visible registers VRA, VRB, VRC. From there, the flow proceeds through multipliers (MPY) that perform parallel concurrent multiplication of the Vector/SIMD data. Adder units comprising carry-skip adders (CSAs) and CPAs may perform additional arithmetic operations. For example, one or more of the CSAs may be used to add in an accumulator value from a vector register file, and one or more of the CPAs may be used to perform a final addition for completion of a multiplication operation, as will be appreciated by those skilled in the art. Computation results are stored in Result registers **464**, and are provided as input operands to the reduction unit **102'**. The reduction unit **102'** sums the input operands in such a way that the summation result produced is the same as that which would be obtained if each operation were executed in series. The reduced sum is stored in the accumulator register file **106'** for further processing.

When performing vector dot products, the MPY blocks perform four multiplies in parallel, the CSA and CPA units perform additional operations or simply pass along the multiplication results for storage in the Result registers **464**, and the reduction unit **102'** sums the multiplication results, along with an accumulator value stored in the accumulator register file **106'**. The result generated by the reduction unit is then stored in the accumulator register file for use in the next iteration, in the manner previously described.

The four parallel multipliers MPY of the vector unit **460** may be viewed as corresponding generally to the multipliers **104** of processor **100** of FIG. **1**.

The accumulator register file **106'** in this example includes a total of sixteen accumulator registers denoted ACC**00** through ACC **15**.

The multithreaded processor **400** may make use of techniques for thread-based access to register files, as described in U.S. Pat. No. 6,904,511, which is commonly assigned herewith and incorporated by reference herein.

FIG. **5** shows an exemplary format for a vector-reduce instruction suitable for execution in the multithreaded processor **400** of FIG. **4**. This instruction is used to specify vector-reduce operations performed by the parallel multipliers and the reduction unit. Such vector-reduce instructions are also referred to herein as vector multiply and reduce operations.

In the figure, OPCODE specifies the operation to be performed, ACCD specifies the accumulator register file location of the accumulator destination register, ACCS specifies the accumulator register file location of the accumulator source register, VRSA specifies the vector register file locations of one set of vector source operands, and VRSB specifies the vector register file locations of the other set of vector source operands.

Using the instruction format shown in FIG. **5**, a SIMD vector processing unit with m parallel multipliers and an (m+1)-input reduction unit can perform a vector-multiply-and-reduce-add (vmulredadd) instruction, which computes

$$ACCD=ACCS+VRSA[1]*VRSB[1]+VSRA[2]*VSRB[2]+\ldots+VSRA[m]*VSRB[m].$$

More specifically, with reference to the exemplary multithreaded processor **400**, this instruction can be executed for m=4 by reading the values corresponding to VSRA[i] and VSRB[i] from the vector register files **462**, using the four parallel multipliers MPY to compute VSRA[i]*VSRB[i], reading ACCS from the accumulator register file **106'**, using the reduction unit **102'** to add the products to ACCS, and writing the result from the reduction unit to back to the accumulator register file, using the address specified by ACCD.

Similarly, a vector-multiply-and-reduce-subtract (vmulredsub) instruction can perform the computation

$$ACCD=ACCS-VRSA[1]*VRSB[1]-VSRA[2]*VSRB[2]-\ldots VSRA[m]*VSRB[m].$$

Each of these vector-reduce instructions can also be performed with saturation after each operation. Other vector-reduce instructions, such as vector-add-reduce-add, which performs the operation

$$ACCD=ACCS+VRSA[1]+VRSB[1]+VSRA[2]+VSRB[2]+\ldots+VSRA[m]+VSRB[m],$$

can also be defined, as will be apparent to those skilled in the art.

FIG. **6** illustrates pipelined execution of two vector-reduce instructions from the same thread, utilizing an instruction format of the type shown in FIG. **5**. In this example, it is assumed without limitation that there are a total of eight threads, and that token triggered threading is used, with round-robin scheduling. The instructions issued by the other threads are not shown in this figure. The pipeline in this example includes 13 stages: instruction fetch (IFE), instruction decode (DEC), read vector register file (RVF), two multiply stages (ML**1** and ML**2**), two adder stages (AD**1** and AD**2**), four reduce stages (RE**1** through RE**4**), result transfer (XFR), and write accumulator file (WAF). In the same cycle with the second adder stage (AD**2**), the processor also reads the accumulator register file (RAF). Thus, a given one of the vector-reduce instructions takes 13 cycles to execute.

It is important to note with regard to this example that if two vector-reduce instruction issue one after the other from the same thread, the first vector-reduce instruction has already written its destination accumulator result back to the accumulator register file (in stage WAF) before the next vector-reduce instruction needs to read its accumulator source register from the register file. Thus two instructions, such as

vmulredadd acc0, acc0, vr1, vr2

vmulredadd acc0, acc0, vr3, vr4

which use the instruction format shown in FIG. **5**, can be issued as consecutive instructions, without causing the processor to stall due to data dependencies. This type of feature can be provided in alternative embodiments using different multithreaded processor, pipeline and reduction unit configurations, as well as different instruction formats.

Another example set of vector multiply and reduce operations will now be described. It should be noted that certain of these operations are similar to or substantially the same as corresponding operations described in the previous examples.

This example set of vector multiply and reduce operations comprises four main types of operations: vector multiply and reduce add with wrap-around (vmredadd), vector multiply and reduce add with saturation (vmredadds), vector multiply and reduce subtract with wrap-around (vmredsub), and vector multiply and reduce subtract with saturation (vmredsubs). These operations take a source accumulator value, ACCS, and two k-element vectors,

$$A=[A[1],A[2],\ldots,A[k]] \text{ and } B=[B[1],B[2],\ldots,B[k]]$$

and compute the value of a destination accumulator ACCD. An important aspect of these operations is that they produce the same result as when all operations are performed in sequential order as individual operations, that is, they provide guaranteed sequential semantics. Generally, the computation result in such an arrangement is exactly the same as that which would be produced using a corresponding sequence of individual instructions, although the invention can be implemented using other types of guaranteed sequential semantics.

The vmredadd operation performs the computation

$$ACCD=\{\ldots\{\{ACCS+\{A[1]*B[1]\}\}+\{A[2]*B[2]\}\}+\ldots+\{A[k]*B[k]\}\},$$

where {T} denotes that T is computed using wrap-around arithmetic. This operation corresponds to multiplying k pairs of vector elements and adding the resulting products to an accumulator in sequential order with wrap-around arithmetic.

The vmredadds operation performs the computation

$$ACCD=<\ldots<<ACCS+<A[1]*B[1]>>+<A[2]*B[2]>>+\ldots+<A[k]*B[k]>>,$$

where <T> denotes that T is computed using saturating arithmetic. This operation corresponds multiplying k pairs of vector elements and adding their products to an accumulator in sequential order, with saturation after each multiplication and each addition.

The vmredsub operation performs the computation

$$ACCD=\{\ldots\{\{ACCS-\{A[1]*B[1]\}\}-\{A[2]*B[2]\}\}-\ldots-\{A[k]*B[k]\}\}.$$

This operation corresponds to multiplying k pairs of vector elements and subtracting the resulting products from an accumulator in sequential order with wrap-around arithmetic.

The vmredsubs operation performs the computation

$$ACCD=<\ldots<<ACCS+<A[1]*B[1]>>+<A[2]*B[2]>>+\ldots+<A[k]*B[k]>>.$$

This operation corresponds to multiplying k pairs of vector elements and subtracting their products from an accumulator in sequential order, with saturation after each multiplication and each subtraction.

Variations of the above operations are also possible based on factors such as the format of the input operands, whether or not rounding is performed, and so on. For example, the above operations can be implemented for operands in unsigned, one's complement, two's complement, or sign-magnitude format. Operands can also be in fixed-point format (in which the number's radix point is fixed) or floating-point format (in which the number's radix point depends on an exponent). Results from operations can either be rounded (using a variety of rounding modes) or kept to full precision.

Further variations of these operations are possible. For example, if ACCS is zero, then the vmredadd and vmredadds instructions correspond to computing the dot product of two k-element vectors with wrap-around or saturating arithmetic. If each element of the A vector is one, then the vmredadd and vmredadds instructions correspond to adding the elements of the B vector to the accumulator with wrap-around or saturating arithmetic.

These example operations are particularly well suited for implementation on a SIMD processor, as described previously herein. With SIMD processing, a single instruction simultaneously performs the same operation on multiple data elements. With the vector multiply and reduce operations described herein, multiplication of the vector elements can be performed in parallel in SIMD fashion, followed by multiple operand additions, with the computational result being the same at that which would be produced is the individual multiplications and additions were performed in sequential order. Providing vector multiply and reduce operations that obtain the same result as when each individual multiplication and addition is performed in sequential order is useful, since this allows code that is developed for one processor to be ported to another processor and still produce the same results.

The format of FIG. **5** can be used to perform the parallel vector multiply and reduce operations of the previous example. As indicated previously, in this figure OPCODE specifies the operation to be performed (e.g., vmredadd, vmredadds, vmredsub, vmredsubs, etc.). ACCD specifies the accumulator register to be used for the destination of result. ACCS specifies the accumulator register to be used for the source accumulator. VRSA specifies the vector register to be used for the k-element source vector, A. VRSB specifies the vector register to be used for the k-element source vector, B. Based on the OPCODE, the elements in the vector registers specified by VSRA and VSRB are multiplied together and the resulting products are added to or subtracted from the accumulator register specified by ACCS, and the result is stored in the accumulator register specified by ACCD.

A number of more specific examples illustrating the performance of the above-described set of vector multiply and reduce operations (vmredadd, vmredadds, vmredsub and vmredsubs) for different input operand values will now be described. In these specific examples, ACCS, ACCD, and all intermediate values are 8-bit two's complement integers, which can take values from −128 to 127. A and B are 4-element vectors (k=4), where each vector element is a 4-bit two's complement integer, which can take values from −8 to 7. When performing wrap-around arithmetic, if a result is greater than 127, its sign bit changes from 0 to 1, which is equivalent to subtracting 256 from the result. If a result is less than −128, its sign bit changes from 1 to 0, which is equivalent to adding 256 to the result. When performing saturating arithmetic, if a result is greater than 127, the result is saturated to 127. If a result is less than −128, it is saturated to −128.

For the vmredadd operation, the addition {113+64} causes the result to wrap around to 113+64−256=−79 and the addition {−114+−36} causes the result to wrap around to −114+−36+256=106.

For the vmredadds operation, the addition <113+64> causes the result to saturate to 127.

For the vmredsub operation, the subtraction {−113−64} causes the result to wrap around to −113−64+256=−79 and the addition {114+36} causes the result to wrap around to 114+36−256=−106.

For the vmredsubs operation, the subtraction <−113−64> causes the result to saturate to −128.

Example 1

ACCS=64,

A[1]=7, A[2]=−8, A[3]=−7, A[4]=−6

B[1]=7, B[2]=−8, B[3]=5, B[4]=6

$$\begin{aligned}
vmredadd:\quad ACCD &= \{\{\{64+\{7*7\}\}+\{-8*-8\}\}+\\
&\quad \{-6*6\}\}+\{-5*7\}\}\\
&= \{\{\{64+49\}+64\}+-35\}+-36\}\\
&= \{\{\{113+64\}+-35\}+-36\}\\
&= \{\{-79+-35\}+-36\}\\
&= \{-114+-36\}\\
&= 106
\end{aligned}$$

$$\begin{aligned}
vmredadds:\quad ACCD &= \langle\langle\langle 64+\langle 7*7\rangle\rangle+\langle -8*-8\rangle\rangle+\\
&\quad \langle -6*6\rangle\rangle+-5*7\rangle\rangle\\
&= \langle\langle\langle 64+49\rangle+64\rangle+-35\rangle+-36\rangle\\
&= \langle\langle 113+64\rangle+-35\rangle+-36\rangle\\
&= \langle\langle 127+-35\rangle+-36\rangle\\
&= \langle 92+-36\rangle\\
&= 56
\end{aligned}$$

Example 2

ACCS=−64,

A[1]=7, A[2]=−8, A[3]=−7, A[4]=−6

B[1]=7, B[2]=−8, B[3]=5, B[4]=6

$$\begin{aligned}
vmredsub:\quad ACCD &= \{\{\{64+\{7*7\}\}-\{-8*-8\}\}-\\
&\quad \{-6*6\}\}-\{-5*7\}\}\\
&= \{\{\{-64-49\}-64\}+35\}-36\}\\
&= \{\{\{-113-64\}+35\}+36\}\\
&= \{\{79+35\}+36\}\\
&= \{114+36\}\\
&= -106
\end{aligned}$$

$$\begin{aligned}
vmredsubs:\quad ACCD &= \langle\langle\langle -64-\langle 7*7\rangle\rangle-\langle -8*-8\rangle\rangle-\\
&\quad \langle -6*6\rangle\rangle-\langle -5*7\rangle\rangle\\
&= \langle\langle\langle -64-49\rangle-64\rangle+35\rangle+36\rangle\\
&= \langle\langle -113-64\rangle+35\rangle+36\rangle\\
&= \langle -128+35\rangle+36\rangle\\
&= \langle -93+36\rangle\\
&= 57
\end{aligned}$$

An advantage of the example set of vector multiply and reduce operations (vmredadd, vmredadds, vmredsub and vmredsubs) described above is that they guarantee sequential semantics. That is, these operations guarantee that the computational result will be the same as that which would be produced using a corresponding sequence of individual instructions.

A wide variety of other types of vector multiply and reduce operations may be implemented using the techniques described herein.

It should be noted that the particular circuitry arrangements shown in FIGS. **1** through **4** are presented by way of illustrative example only, and additional or alternative elements not explicitly shown may be included, as will be apparent to those skilled in the art.

It should also be emphasized that the present invention does not require the particular multithreaded processor configuration shown in FIG. **4**. The invention can be imple-

mented in a wide variety of other multithreaded or non-multithreaded processor configurations.

Thus, the above-described embodiments of the invention are intended to be illustrative only, and numerous alternative embodiments within the scope of the appended claims will be apparent to those skilled in the art. For example, the particular arithmetic unit, reduction unit and accumulator unit configurations shown may be altered in other embodiments. Also, as noted above, pipeline configurations, threading types and instruction formats may be varied to accommodate the particular needs of a given application.

What is claimed is:

1. A multi-threaded vector processor comprising:

a plurality of vector arithmetic units for performing parallel concurrent vector operations on vectors comprising vector elements;

a vector accumulator unit; and

a vector reduction unit coupled between the plurality of vector arithmetic units and the vector accumulator unit, the vector reduction unit receiving products of vector elements from the vector arithmetic units and a first accumulator value from the vector accumulator unit;

wherein the vector reduction unit is pipelined and operative to process the products and the first accumulator value, and to generate a second accumulator value for delivery to the vector accumulator unit;

wherein the multi-threaded vector processor implements a plurality of vector multiply and reduce instructions having guaranteed sequential semantics such that computation results of a vector multiply and reduce instruction is the same as that which is produced using a corresponding sequence of individual instructions; and

wherein a vector multiply and reduce instruction computed for a given thread is executed concurrently with operations from other threads, the number of cycles between execution of the vector multiply and reduce instruction from the given thread being greater than or equal to a number of pipeline stages in the vector reduction unit plus any additional cycles needed to write to and read from the vector accumulator unit.

2. The processor of claim **1** wherein the vector multiply and reduce add with wrap-around performs the computation:

$$ACCD=\{\ldots\{\{ACCS+\{A[1]*B[1]\}\}+\{A[2]*B[2]\}\}+\ldots+\{A[k]*B[k]\}\},$$

where A and B are k-bit input vectors, ACCS denotes the first accumulator value, ACCD denotes the second accumulator value, and {T} denotes that T is computed using wrap-around arithmetic.

3. The processor of claim **1** wherein the vector multiply and reduce add with saturation performs the computation:

$$ACCD=<\ldots<<ACCS+<A[1]*B[1]>>+<A[2]*B[2]>>+\ldots+<A[k]*B[k]>>,$$

where A and B are k-bit input vectors, ACCS denotes the first accumulator value, ACCD denotes the second accumulator value, and <T> denotes that T is computed using saturating arithmetic.

4. The processor of claim **1** wherein the plurality of vector multiply and reduce operations comprises a vector multiply and reduce subtract with wrap-around which multiplies pairs of vector elements and subtracts the resulting products from the first accumulator value in sequential order with wrap-around arithmetic.

5. The processor of claim **4** wherein the vector multiply and reduce subtract with wrap-around performs the computation:

$$ACCD=\{\ldots\{\{ACCS-\{A[1]*B[1]\}\}-\{A[2]*B[2]\}\}-\ldots-\{A[k]*B[k]\}\},$$

where A and B are k-bit input vectors, ACCS denotes the first accumulator value, ACCD denotes the second accumulator value, and {T} denotes that T is computed using wrap-around arithmetic.

6. The processor of claim **1** wherein the plurality of vector multiply and reduce operations comprises a vector multiply and reduce subtract with saturation which multiplies pairs of vector elements and subtracts the resulting products from the first accumulator value in sequential order with saturation after each multiplication and each subtraction.

7. The processor of claim **6** wherein the vector multiply and reduce subtract with saturation performs the computation:

$$ACCD=<\ldots<<ACCS-<A[1]*B[1]>>-<A[2]*B[2]>>-\ldots-<A[k]*B[k]>>,$$

where A and B are k-bit input vectors, ACCS denotes the first accumulator value, ACCD denotes the second accumulator value, and <T> denotes that T is computed using saturating arithmetic.

8. The processor of claim **1** wherein input vectors to which a given one of the vector multiply and reduce operations is applied are in one of an unsigned format, a one's complement format, a two's complement format, and a sign-magnitude format.

9. The processor of claim **1** wherein input vectors to which a given one of the vector multiply and reduce operations is applied are in one of a fixed-point format and a floating-point format.

10. The processor of claim **1** wherein results of a given one of the vector multiply and reduce operations are rounded.

11. The processor of claim **1** wherein results of a given one of the vector multiply and reduce operations are maintained at full precision.

12. The processor of claim **1** wherein the first accumulator value is zero, and a given one of the vector multiply and reduce operations comprises a dot product.

13. The processor of claim **1** wherein each element of a first input vector has a value of one, and a given one of the vector multiply and reduce operations comprises adding elements of a second input vector to the first accumulator value.

14. The processor of claim **1** wherein the processor comprises a single instruction multiple data (SIMD) processor, a given one of the vector multiply and reduce operations performing parallel multiplications of vector elements.

15. The processor of claim **1** wherein the plurality of arithmetic units comprises a plurality of multipliers arranged in parallel with one another.

16. The processor of claim **1** wherein the reduction unit is configured to provide controllable selection between at least a first type of computation with saturation after each of a plurality of addition or subtraction operations and a second type of computation with wrapping around of results of the addition or subtraction operations, responsive to an applied control signal.

17. The processor of claim **1**, wherein the plurality of vector multiply and reduce instructions comprises a vector multiply and reduce add with wrap-around which multiplies pairs of vector elements and adds the resulting products to the first accumulator value with wrap-around after each multiplication and each addition and a reduce add with saturation which multiplies pairs of vector elements and adds the resulting products to the first accumulator value with saturation after each multiplication and each addition, wherein the order of adding the resulting products to the first accumulator value and selection by the vector processor of one of wrap around

arithmetic and saturation arithmetic are specified by the type of vector multiply and reduce instruction.

18. An integrated circuit comprising at least one multi-threaded vector processor, the multi-threaded vector processor comprising:

a plurality of vector arithmetic units for performing parallel concurrent vector operations on vectors comprising vector elements;

a vector accumulator unit; and

a vector reduction unit coupled between the plurality of vector arithmetic units and the vector accumulator unit, the vector reduction unit being pipelined and configured to receive products of vector elements from the vector arithmetic units and a first accumulator value from the vector accumulator unit;

wherein the vector reduction unit is operative to process the products and the first accumulator value, and to generate a second accumulator value for delivery to the vector accumulator unit;

wherein the multi-threaded vector processor implements a plurality of vector multiply and reduce instructions having guaranteed sequential semantics such that computation results of a vector multiply and reduce instruction is the same as that which is produced using a corresponding sequence of individual instructions; and

wherein a vector multiply and reduce instruction computed for a given thread is executed concurrently with operations from other threads, the number of cycles between execution of the vector multiply and reduce instruction from the given thread being greater than or equal to a number of pipeline stages in the vector reduction unit plus any additional cycles needed to write to and read from the vector accumulator unit.

19. An apparatus for use in a multi-threaded vector processor comprising a plurality of vector arithmetic units for performing parallel concurrent vector operations on vectors comprising vector elements and a vector accumulator unit, the apparatus comprising:

a vector reduction unit coupled between the plurality of vector arithmetic units and the vector accumulator unit, the vector reduction unit being configured to receive products of vector elements from the arithmetic units and a first accumulator value from the vector accumulator unit;

wherein the vector reduction unit is pipelined and operative to process the products and the first accumulator value, and to generate a second accumulator value for delivery to the vector accumulator unit;

wherein the multi-threaded vector processor implements a plurality of vector multiply and reduce instructions having guaranteed sequential semantics such that computation results of a vector multiply and reduce instruction is the same as that which is produced using a corresponding sequence of individual instructions; and

wherein a vector multiply and reduce instruction computed for a given thread is executed concurrently with operations from other threads, the number of cycles between execution of the vector multiply and reduce instruction from the given thread being greater than or equal to a number of pipeline stages in the vector reduction unit plus any additional cycles needed to write to and read from the vector accumulator unit.

20. The apparatus of claim **19**, wherein the plurality of vector multiply and reduce instructions comprises a vector multiply and reduce add with wrap-around which multiplies pairs of vector elements and adds the resulting products to the first accumulator value with wrap-around after each multiplication and each addition and a reduce add with saturation which multiplies pairs of vector elements and adds the resulting products to the first accumulator value with saturation after each multiplication and each addition, wherein the order of adding the resulting products to the first accumulator value and selection by the vector processor of one of wrap around arithmetic and saturation arithmetic are specified by the type of vector multiply and reduce instruction.

21. The integrated circuit of claim **18**, wherein the plurality of vector multiply and reduce instructions comprises a vector multiply and reduce add with wrap-around which multiplies pairs of vector elements and adds the resulting products to the first accumulator value with wrap-around after each multiplication and each addition and a reduce add with saturation which multiplies pairs of vector elements and adds the resulting products to the first accumulator value with saturation after each multiplication and each addition, wherein the order of adding the resulting products to the first accumulator value and selection by the vector processor of one of wrap around arithmetic and saturation arithmetic are specified by the type of vector multiply and reduce instruction.

* * * * *